ECOLE DES MINES DE SAINT ETIENNE
CAMPUS G. CHARPAK PROVENCE

RAPPORT

Digital Systems Design - ASCON Project

Gevorg ISHKHANYAN

15 septembre 2024

Table des matières

1 Introduction

Authenticated encryption with associated data (AEAD) is widely used today in network protocols to secure communications. It ensures both the confidentiality and authenticity of exchanged data. For this, the message content is encrypted while the header remains in plaintext. It includes information such as the source and destination IP addresses, as well as the message size, allowing proper packet routing in the network. The ASCON128 algorithm guarantees confidentiality by encrypting the message content and also ensures the authenticity of the header using a tag. The recipient must verify this tag ; if it does not match the expected value, the message is rejected.

An image that represents this encryption :

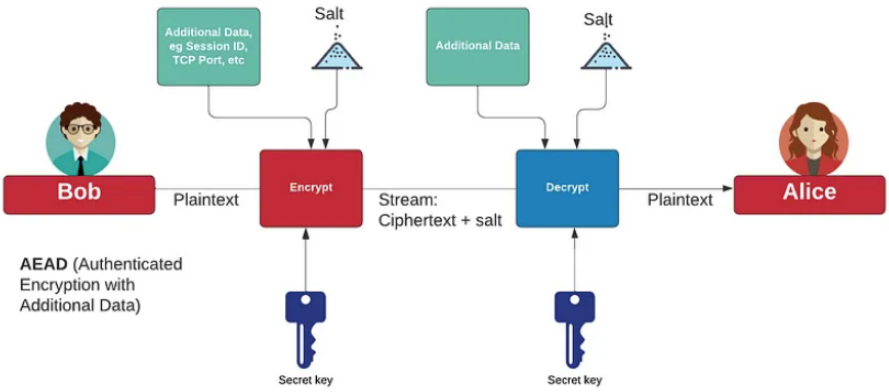

Figure 1 – AEAD Encryption

In this project, I studied in detail the architecture of the ASCON128 algorithm, explaining the various stages of the encryption process, such as state initialization, processing associated data, encrypting the plaintext, and generating the authentication tag. I also examined the specifications of the permutations, as well as the constant addition, substitution, and linear diffusion layers that make them up. Unfortunately, I was not able to complete the entire project. Nevertheless, I will show the work I have done and what I planned to do next.

This report will present in detail the different phases of the project, the design choices made, the results of the simulations, as well as any encountered obstacles and the proposed solutions. The objective of this project was to understand the ASCON128 system, including all the modules it comprises, to design and test these different modules, and to discover and become familiar with a hardware description language (HDL) for the first time, which for this project is SystemVerilog.

To complete this project, I will use the Modelsim software to compile the code and simulate, as well as VSCode to edit my code.

2 The ASCON Project

2.1 System Operation

In this section, I will explain the operation of the ASCON system as a whole. I will discuss the modules in detail in the following sections. In the rest of this report, the term "to XOR" or "XORing" will be used as a verb to indicate the use of the XOR (exclusive OR) function.
This version of the ASCON128 system is a deliberately simplified version of the original algorithm to allow for its implementation within a limited time frame. The remainder of the report takes this simplification into account.

2.1.1 Notations

We will use the following terminology and notations :
— The algorithm operates on a current state (or state S) of 320 bits.
— This state is updated with an operation called permutation. The permutation comprises either 6 iterations or 12 iterations and will be noted as p6 (respectively p12). The iterations are also called rounds.
— The state is divided into two parts :
  — An external part of r = 64 bits, noted $S_r = S_{64}$
  — An internal part of c = 256 bits, noted $S_c = S_{256}$

The state is thus the concatenation of the two parts, and S = $\{S_{64}, S_{256}\}\} = \{S_r, S_c\}$. The state S of 320 bits is divided into 5 register arrays x_i of 64 bits each, as represented in Figure 2.
(This image comes from the project assignment)

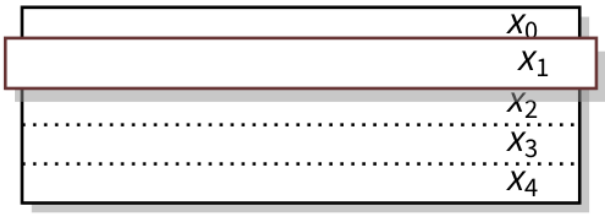

FIGURE 2 – Representation of S = $\{x_0, x_1, x_2, x_3, x_4\}$

The state can also be interpreted as 64 columns of 5 bits each.
Additional notations :

— K : Secret key K of 128 bits
— N : Arbitrary number nonce N of 128 bits
— T : Tag T of 128 bits
— P : Plaintext P of 184 bits
— C : Ciphertext C of 184 bits
— A : Associated data A of 48 bits
— IV : Initialization vector IV of 64 bits 0x80400C0600000000

2.1.2 Encryption Steps

Encryption takes place in several stages :
(This image comes from the project assignment)

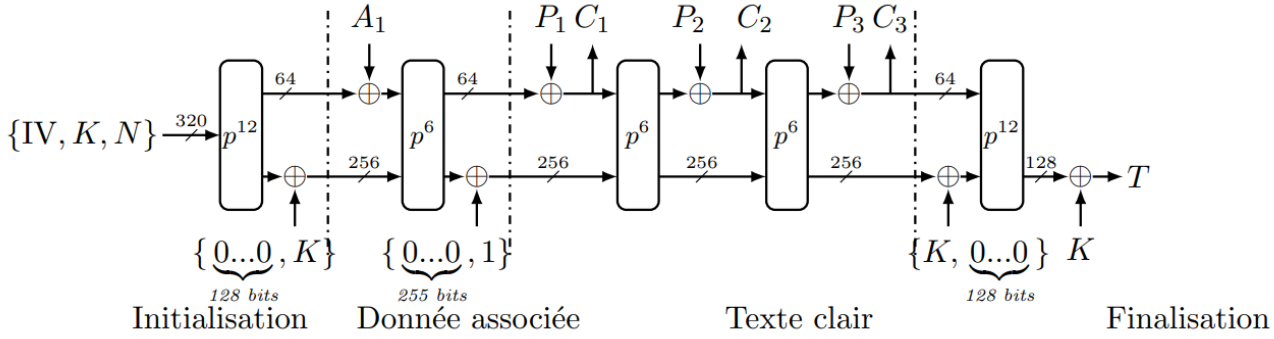

FIGURE 3 – The encryption steps

The initialization and associated data phases are the two parts where we prepare the data for encryption :

- Initialization : We input {IV, K, N} to perform 12 permutations, then XOR (x3, x4) with the key K (xor_key).
- Associated data : We XOR the first 64 bits of the initialization output with the associated data A1, then after performing 6 permutations, we XOR the least significant bit (lsb) of the output with 1.

The Plaintext phase is where the overall message encryption takes place. This encryption is divided into 3 smaller encryptions during which the parts of the message (Plaintext P) are XORed with the associated data, and then the encrypted message (Ciphertext C) is returned. The finalization phase serves to communicate the tag to the other party for message decryption.

- Plaintext : We XOR the first 64 bits with P1, then return this result as the ciphertext.
- Finalization : We XOR the last 256 bits (x1, x2, x3, x4) with {K,0...0}, then after performing 12 permutations, we XOR the last 128 bits (x3, x4) with the key K.

2.1.3 The Complete ASCON128 System

The ASCON128 system (also called ASCON_Top) is composed of several blocks.

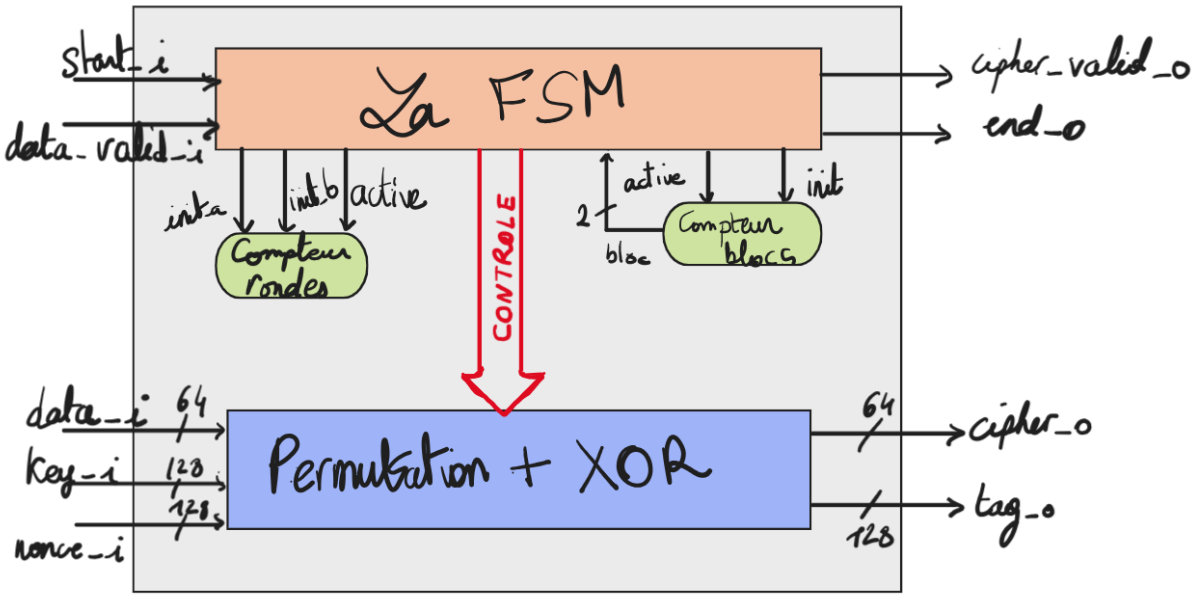

FIGURE 4 – Architecture of the ASCON128 system

Not forgetting the resetb_i and clock_i signals as inputs.

We can relate this to the processor architecture course. Indeed, because the FSM (Finite State Machine) plays the role of the "Control Path," while the Permutation and XORs play the role of the "Data Path." This connection with the course gave me a different perspective, which helped me better understand the system's operation. I will explain their functions in the next sections.

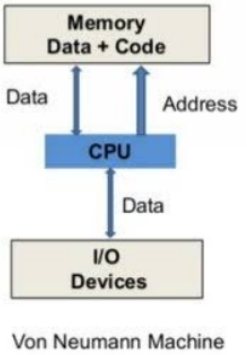

FIGURE 5 – Von Neuman Architecture

2.2 Work Organization

To ensure better organization and thus easily navigate through the files, the codes were classified into different directories as it is important to separate compiled files from source codes and testbenches. To achieve this, I organized the files as follows :

- The file compil_ascon.txt is used for compilation.
- The source files are in SRC/RTL.
- The testbench files are in SRC/BENCH.
- The executable files and compiled libraries are in LIB/LIB_RTL and LIB/LIB_BENCH.
- The rest of the files below compil_ascon.txt will not be used, as I am using ModelSim locally and not from an external server (such as *tallinn*, for example).

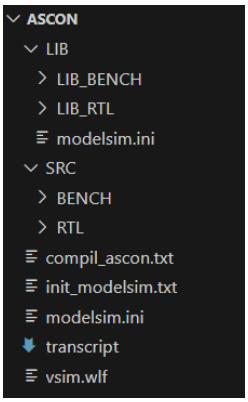

FIGURE 6 – File code organization

2.3 Compilation and Simulation

The compilation is done using the compilation script located in compil_ascon.txt. When Modelsim is launched, we find ourselves in the following folder :

```
ModelSim> pwd
# C:/intelFPGA/20.1
```

FIGURE 7 – Starting folder in ModelSim

Then we need to uncomment the testbench that we want to compile and simulate, and run the following commands in the ModelSim terminal :

```
# Go to the ASCON folder, for me on Windows it will be:
cd ../../Users/"user"/Desktop/ASCON

# Then run the compilation script
source compil_ascon.txt
```

Then observe the simulation results.

3 Project Components

The main components of the ASCON128 algorithm are the two permutations p^6 and p^{12} of the 320-bit state S. These two permutations iteratively apply the round transformation p, which consists of three stages P_C, P_S, and P_L, such that :
$P = P_C \circ P_S \circ P_L$. The permutations p^6 and p^{12} differ only in the number of rounds.

Here is a diagram of the complete permutation. This version also includes the initial and final XOR operations, as well as the registers for the ciphers and the tag :

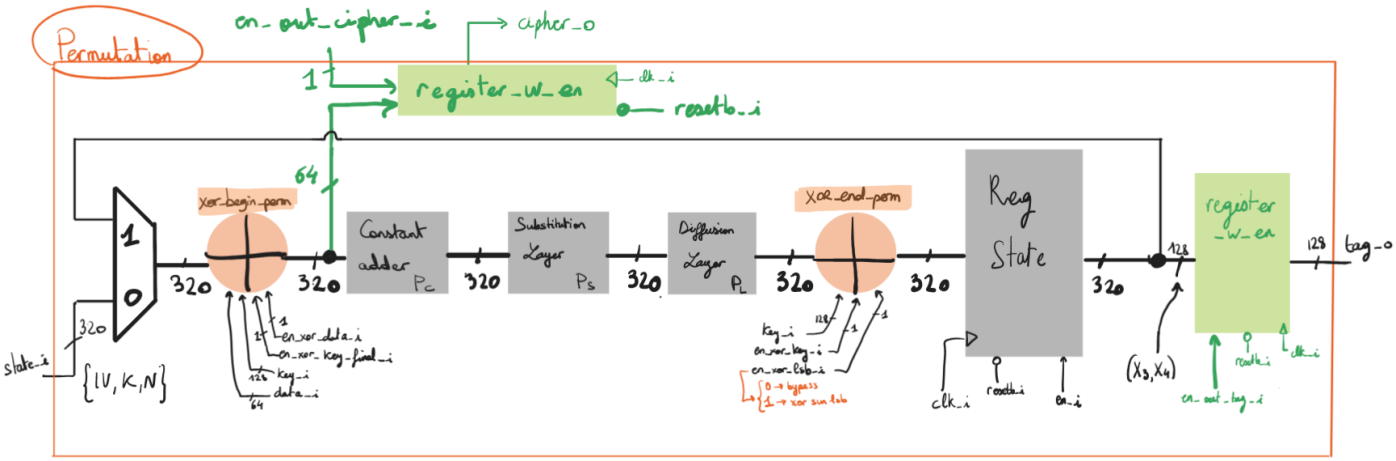

FIGURE 8 – A complete permutation

3.1 Constant Addition

The constant addition P_C adds a round constant c_r to the x_2 register of the state S during round i, such that $x_2 \longleftarrow x_2 \bigoplus c_r$. The constants are defined in the appendix, in Figure 16. You need to add the constants to the ascon_pack package.

```
1     localparam logic [7:0]   round_constant [0:11] = {8'hF0, 8'hE1, 8'
         hD2, 8'hC3, 8'hB4, 8'hA5, 8'h96, 8'h87, 8'h78, 8'h69, 8'h5A, 8'
         h4B};
```

Listing 1 – Adding constants to ascon_pack.sv

We notice that the *state_o* is indeed modified as expected compared to *state_i*.

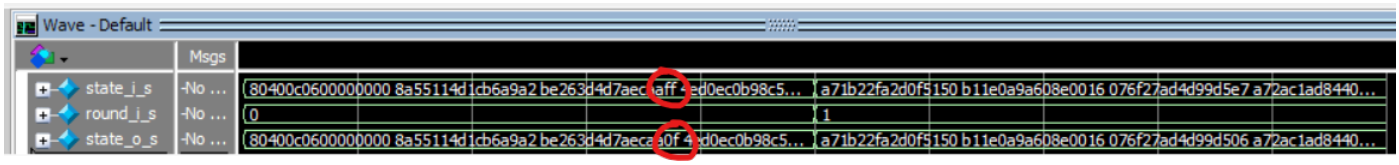

FIGURE 9 – Constant addition chronogram from constant_addition_tb.sv

3.2 Substitution Layer

The substitution layer P_S modifies the state S by applying in parallel the 5-bit column substitution using the S-box substitution table defined in the appendix in Figure 17. The state S is interpreted in this case as a table of 64 columns, as mentioned in the notations section. The S-box table needs to be added to the ascon_pack package. First, you need to create the module ascon_sbox.sv to apply the function $S(x)$, then create the module substitution_layer.sv.

```
localparam logic [4:0]   sbox_t [0:31] = {5'h04, 5'h0B, 5'h1F, 5'h14
    , 5'h1A, 5'h15, 5'h09, 5'h02, 5'h1B, 5'h05, 5'h08, 5'h12, 5'h1D,
    5'h03, 5'h06, 5'h1C, 5'h1E, 5'h13, 5'h07, 5'h0E, 5'h00, 5'h0D, 5'
    h11, 5'h18, 5'h10, 5'h0C, 5'h01, 5'h19, 5'h16, 5'h0A, 5'h0F, 5'
    h17 };
```

Listing 2 – Adding the S-box table to ascon_pack.sv

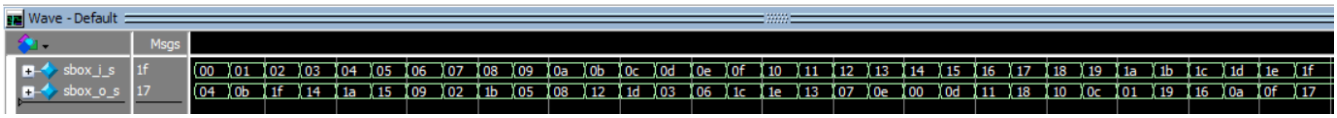

FIGURE 10 – Chronogram of ascon_sbox_tb.sv

We notice that the *sbox_o_s* is correctly modified as expected compared to the *sbox_i_s*, and the result matches the S-box table.

Naturally, the substitution layer works correctly according to the testbench since the S-box is fully functional.

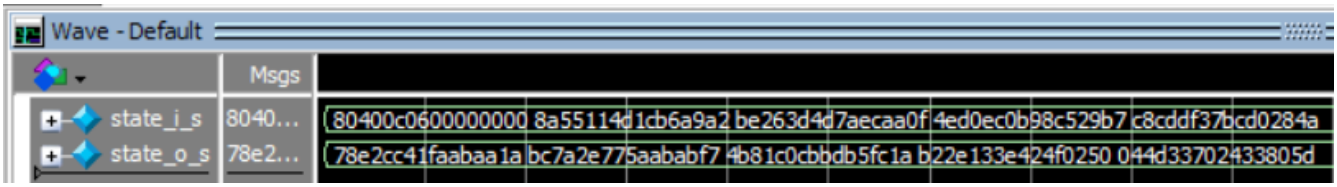

FIGURE 11 – Chronogram of substitution_layer_tb.sv

3.3 Diffusion Layer

(This image comes from the project assignment)

La couche de diffusion linéaire p_L applique une diffusion à chacune des lignes de 64 bits ou registres x_i. On applique ainsi l'opération suivante :

$$x_i \leftarrow \Sigma_i(x_i)$$

$$\begin{array}{lllllllllllll}
x_0 & \leftarrow & \Sigma_0(x_0) & = & x_0 & \oplus & (x_0 & \ggg & 19) & \oplus & (x_0 & \ggg & 28) \\
x_1 & \leftarrow & \Sigma_1(x_1) & = & x_1 & \oplus & (x_1 & \ggg & 61) & \oplus & (x_1 & \ggg & 39) \\
x_2 & \leftarrow & \Sigma_2(x_2) & = & x_2 & \oplus & (x_2 & \ggg & 1) & \oplus & (x_2 & \ggg & 6) \\
x_3 & \leftarrow & \Sigma_3(x_3) & = & x_3 & \oplus & (x_3 & \ggg & 10) & \oplus & (x_3 & \ggg & 17) \\
x_4 & \leftarrow & \Sigma_4(x_4) & = & x_4 & \oplus & (x_4 & \ggg & 7) & \oplus & (x_4 & \ggg & 41)
\end{array}$$

(This image comes from the project assignment)

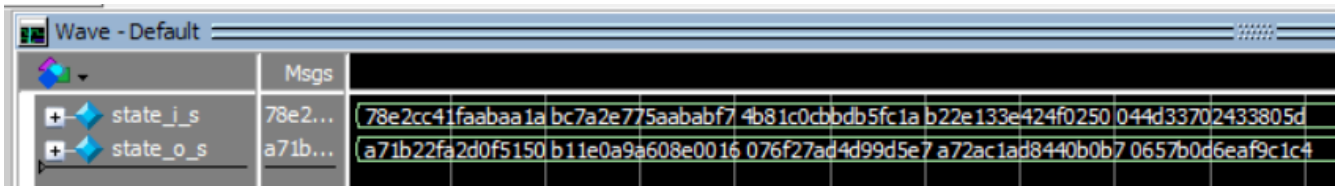

FIGURE 12 – Chronogram of diffusion_layer_tb.sv

We notice that the *state_o_s* is correctly modified as expected compared to the *state_i_s*, and the result matches the expected outcome.

3.4 The Permutation

I first implemented a permutation_step_1.sv to understand how the permutation works. Then I added the initial and final XORs, as well as the registers for the ciphers and the tag. I will explain these additions later.

The expected values for the permutation (here for the 11th permutation, for example) are the following :

```
1    9de1e2d04b50bd86 13141c888a702ce4 55aa22c50d252d83 6530ee4949ecd5b1
       1209bee19f4a7ade
```

Listing 3 – Expected value of state_o at the 11th permutation

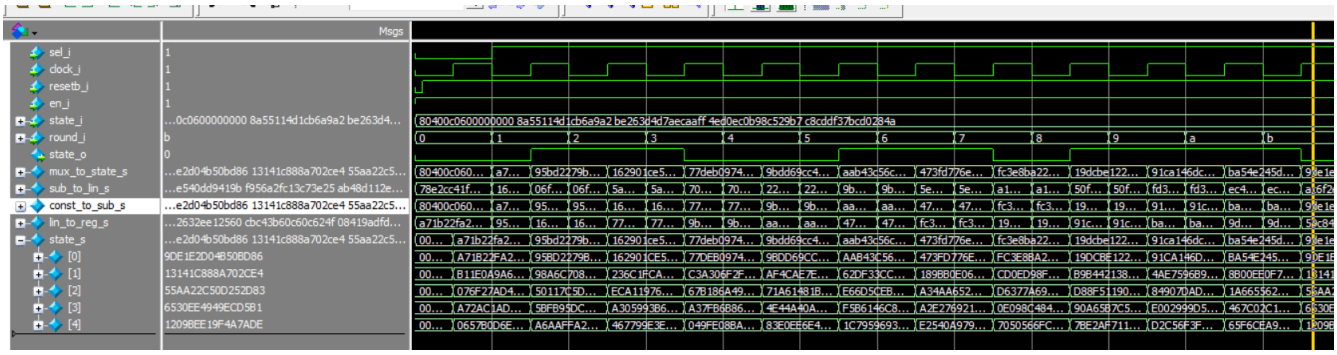

FIGURE 13 – Chronogram of permutation_step_1_tb.sv

We have exactly the same values as expected, so the permutation works correctly by itself.

3.5 The Permutation with XOR and Added Registers

The XOR begin block is used during the Associated Data, Plaintext, and Finalization phases. This module performs an XOR operation between the input signal from the multiplexer and the key K or the associated data. If en_xor_data_i is activated, the operation is done with the data, and if en_xor_key_i is activated, the operation is done with K.

Moreover, the Initialization, Associated Data, and Finalization phases use the XOR end block. This module works the same way as the XOR begin, except that it performs the operation between the signal from the multiplexer and the key or the least significant bit of x_4 with a bit 1. The activation signals are en_xor_end_key_i and en_xor_lsb_i, respectively.

To properly implement the permutation with the XORs and the registers, it is necessary to declare internal wires that will connect the modules together, using the same wire for the output of one module and the input of another.

```
1       // Internal net declaration
2 type_state mux_to_xor_begin_s, xor_begin_to_add_s,
3  const_to_sub_s, sub_to_lin_s, lin_to_xor_end_s, xor_end_to_reg_s;
```

Listing 4 – For making internal connections

After adding the XORs and registers, we rerun the simulation for a permutation.

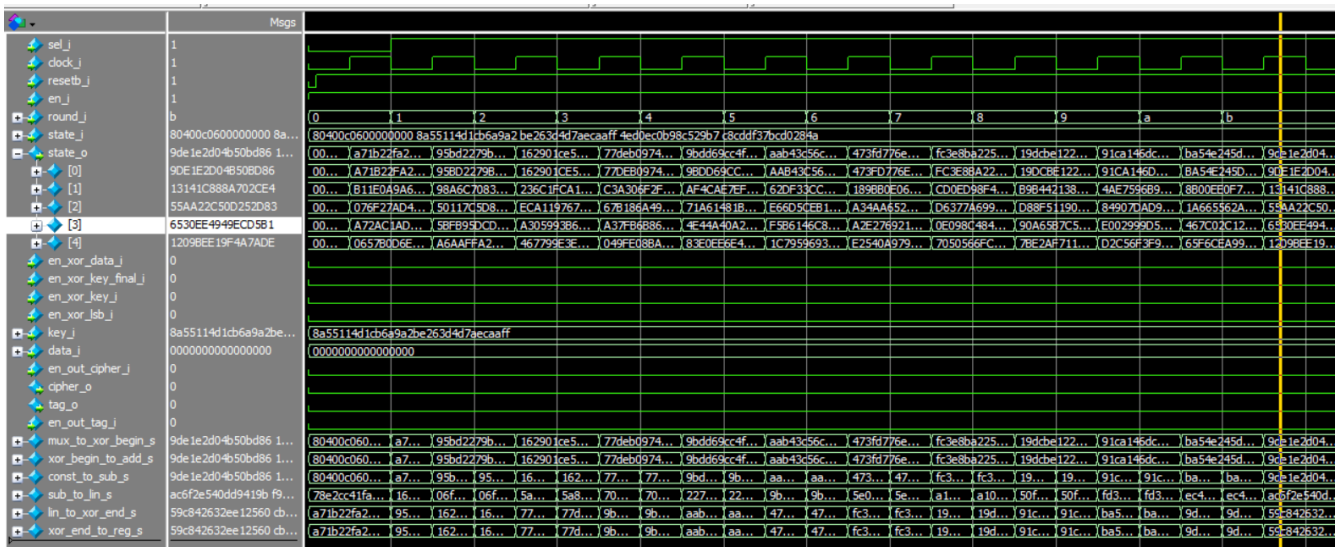

FIGURE 14 – Chronogram of permutation_tb.sv

We have exactly the same values as expected, so the complete permutation works.

3.6 The FSM

I was unable to complete the FSM for the entire encryption process (Initialization, Associated Data, Plaintext, and Finalization), but I did implement the initialization part of the encryption. This FSM is a Moore machine in our case, as the future states depend only on the previous states.

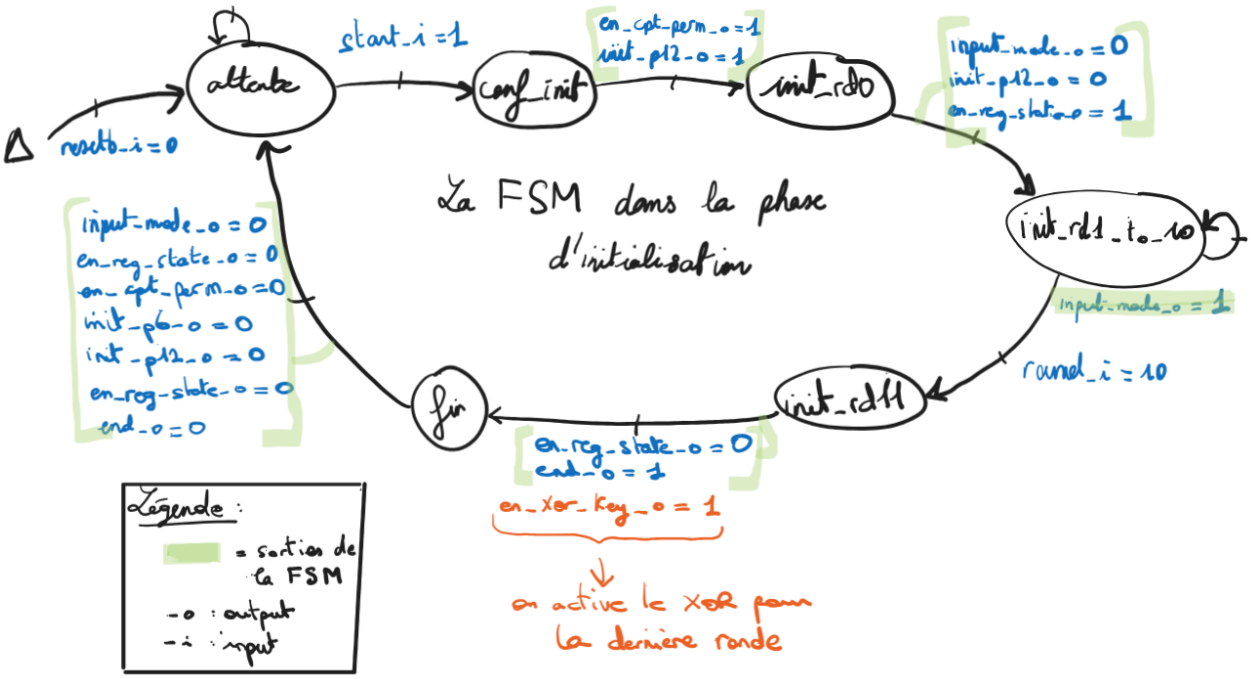

FIGURE 15 – Initialization of the FSM

4 Conclusion

To conclude, this Digital Systems Design project on ASCON128 was a very stimulating experience. I successfully implemented the complete permutation with its testbench to verify its proper operation.

The project was not fully completed, so the next step would naturally be to finalize the global FSM (not just for the initialization phase), as well as the ASCON_TOP part to link all these modules together. However, the hardest part has been accomplished with the permutation and the implemented registers and XORs. In addition to finalizing the FSM, it would be interesting to try optimizing performance by using a Mealy machine instead of a Moore machine (again, making a connection to the processor architecture course), but of course, this solution would need to be compatible with our reliability requirements, particularly from an electrical standpoint.

To conclude, this project allowed me to acquire valuable knowledge and skills in the field of digital systems design. I am very happy to have been able to complete this project in my first year of the ISMIN program, as this change will already give me useful skills

for a potential second-year internship. It also served as an introduction to cryptography, something I had never experimented with before.

I would like to express my gratitude to all those who taught me a lot on this subject, particularly my professor **Jean-Max DUTERTRE**, and also my classmate **Ibrahim HADJ-ARAB**, who occasionally helped me with some debugging.

5 Resources / Appendix

Figure 1 : https ://medium.com/asecuritysite-when-bob-met-alice/asconrust-and-aead-b237afb78a83

Figure 4 : https ://www.quora.com/Which-is-the-most-commonly-usedcomputer-architecture-von-Nuemann-or-Harvard-architecture-and-why

Table of constants for constant addition :

Ronde r de p^{12}	Ronde r de p^6	Constante c_r
0		00000000000000000f0
1		00000000000000000e1
2		00000000000000000d2
3		00000000000000000c3
4		00000000000000000b4
5		00000000000000000a5
6	0	0000000000000000096
7	1	0000000000000000087
8	2	0000000000000000078
9	3	0000000000000000069
10	4	000000000000000005a
11	5	000000000000000004b

FIGURE 16 – Constant addition, (This image comes from the project assignment)

x	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
$S(x)$	04	0B	1F	14	1A	15	09	02	1B	05	08	12	1D	03	06	1C	1E	13	07	0E	00	0D	11	18	10	0C	01	19	16	0A	0F	17

FIGURE 17 – S-box substitution table, (This image comes from the project assignment)